Start by importing necessary packages

You will begin by importing necessary libraries for this notebook. Run the cell below to do so.

PyTorch and Intro to Training

```
!pip install thop
import math
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import thop
import matplotlib.pyplot as plt
from tqdm import tqdm
import time
```

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7 kB)
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (from thop) (2.5.1+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from torch->thop) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (3.1.5)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (from torch->thop) (2024.10.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (12
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.3.1
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (11.0.2.5
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (10.3.2
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (11.4
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: triton==3.1.0 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (3.1.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packages (from torch->thop) (1.13.1)
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.11/dist-packages (from nvidia-cusolver-cu12==11.4.5.
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch->thop) (1.3
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->torch->thop) (3.0.2)
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Installing collected packages: thop
Successfully installed thop-0.1.1.post2209072238
```

Checking the torch version and CUDA access

Let's start off by checking the current torch version, and whether you have CUDA availablity.

```
print("torch is using version:", torch.__version__, "with CUDA=", torch.cuda.is_available())
```

```
torch is using version: 2.5.1+cu121 with CUDA= True
```

By default, you will see CUDA = False, meaning that the Colab session does not have access to a GPU. To remedy this, click the Runtime menu on top and select "Change Runtime Type", then select "T4 GPU".

Re-run the import cell above, and the CUDA version / check. It should show now CUDA = True

Sometimes in Colab you get a message that your Session has crashed, if that happens you need to go to the Runtime menu on top and select "Restart session".

You won't be using the GPU just yet, but this prepares the instance for when you will.

Please note that the GPU is a scarce resource which may not be available at all time. Additionally, there are also usage limits that you may run into (although not likely for this assignment). When that happens you need to try again later/next day/different time of the day. Another reason to start the assignment early!

A Brief Introduction to PyTorch

PyTorch, or torch, is a machine learning framework developed my Facebook AI Research, which competes with TensorFlow, JAX, Caffe and others.

Roughly speaking, these frameworks can be split into dynamic and static defintion frameworks.

Static Network Definition: The architecture and computation flow are defined simultaneously. The order and manner in which data flows through the layers are fixed upon definition. These frameworks also tend to declare parameter shapes implicitly via the compute graph. This is typical of TensorFlow and JAX.

Dynamic Network Definition: The architecture (layers/modules) is defined independently of the computation flow, often during the object's initialization. This allows for dynamic computation graphs where the flow of data can change during runtime based on conditions. Since the network exists independent of the compute graph, the parameter shapes must be declared explitly. PyTorch follows this approach.

All ML frameworks support automatic differentiation, which is necessary to train a model (i.e. perform back propagation).

Let's consider a typical pytorch module. Such modules will inherit from the torch.nn.Module class, which provides many built in functions such as a wrapper for `__call__`, operations to move the module between devices (e.g. `cuda()`, `cpu()`), data-type conversion (e.g. `half()`, `float()`), and parameter and child management (e.g. `state_dict()`, `parameters()`).

```
# inherit from torch.nn.Module
class MyModule(nn.Module):
  # constructor called upon creation
  def __init__(self):
    # the module has to initialize the parent first, which is what sets up the wrapper behavior
    super().__init__()

    # we can add sub-modules and parameters by assigning them to self
    self.my_param = nn.Parameter(torch.zeros(4,8)) # this is how you define a raw parameter of shape 4x5
    self.my_sub_module = nn.Linear(8,12)      # this is how you define a linear layer (tensorflow calls them Dense) of shape 8x12

    # we can also add lists of modules, for example, the sequential layer
    self.net = nn.Sequential(  # this layer type takes in a collection of modules rather than a list
        nn.Linear(4,4),
        nn.Linear(4,8),
        nn.Linear(8,12)
    )

    # the above when calling self.net(x), will execute each module in the order they appear in a list
    # it would be equivelent to x = self.net[2](self.net[1](self.net[0](x)))

    # you can also create a list that doesn't execute
    self.net_list = nn.ModuleList([
        nn.Linear(7,7),
        nn.Linear(7,9),
        nn.Linear(9,14)
    ])

    # sometimes you will also see constant variables added to the module post init
    foo = torch.Tensor([4])
    self.register_buffer('foo', foo) # buffers allow .to(device, type) to apply

  # let's define a forward function, which gets executed when calling the module, and defines the forward compute graph
  def forward(self, x):

    # if x is of shape Bx4
    h1 =  x @ self.my_param # tensor-tensor multiplication uses the @ symbol
    # then h1 is now shape Bx8, because my_param is 4x8... 2x4 * 4x8 = 2x8

    h1 = self.my_sub_module(h1) # you execute a sub-module by calling it
    # now, h1 is of shape Bx12, because my_sub_module was a 8x12 matrix

    h2 = self.net(x)
    # similarly, h2 is of shape Bx12, because that's the output of the sequence
    # Bx4 -(4x4)-> Bx4 -(4x8)-> Bx8 -(8x12)-> Bx12

    # since h1 and h2 are the same shape, they can be added together element-wise
    return h1 + h2
```

Then you can instantiate the module and perform a forward pass by calling it.

```
# create the module
module = MyModule()

# you can print the module to get a high-level summary of it
print("=== printing the module ===")
print(module)
print()
# notice that the sub-module name is in parenthesis, and so are the list indicies
# let's view the shape of one of the weight tensors
print("my_sub_module weight tensor shape=", module.my_sub_module.weight.shape)
# the above works because nn.Linear has a member called .weight and .bias
# to view the shape of my_param, you would use module.my_param
# and to view the shape of the 2nd elment in net_list, you would use module.net_list[1].weight

# you can iterate through all of the parameters via the state dict
print()
print("=== Listing parameters from the state_dict ===")
for key,value in module.state_dict().items():
  print(f"{key}: {value.shape}")
```

```
=== printing the module ===
MyModule(
  (my_sub_module): Linear(in_features=8, out_features=12, bias=True)
  (net): Sequential(
    (0): Linear(in_features=4, out_features=4, bias=True)
    (1): Linear(in_features=4, out_features=8, bias=True)
    (2): Linear(in_features=8, out_features=12, bias=True)
  )
  (net_list): ModuleList(
    (0): Linear(in_features=7, out_features=7, bias=True)
    (1): Linear(in_features=7, out_features=9, bias=True)
    (2): Linear(in_features=9, out_features=14, bias=True)
  )
)

my_sub_module weight tensor shape= torch.Size([12, 8])

=== Listing parameters from the state_dict ===
my_param: torch.Size([4, 8])
foo: torch.Size([1])
my_sub_module.weight: torch.Size([12, 8])
my_sub_module.bias: torch.Size([12])
net.0.weight: torch.Size([4, 4])
net.0.bias: torch.Size([4])
net.1.weight: torch.Size([8, 4])
net.1.bias: torch.Size([8])
net.2.weight: torch.Size([12, 8])
net.2.bias: torch.Size([12])
net_list.0.weight: torch.Size([7, 7])
net_list.0.bias: torch.Size([7])
net_list.1.weight: torch.Size([9, 7])
net_list.1.bias: torch.Size([9])
net_list.2.weight: torch.Size([14, 9])
net_list.2.bias: torch.Size([14])
```

```
# you can perform a forward pass by first creating a tensor to send through
x = torch.zeros(2,4)
# then you call the module (this invokes MyModule.forward() )
y = module(x)

# then you can print the result and shape
print(y, y.shape)
```

```
tensor([[ 0.5492,  0.6238, -0.4663,  0.3776, -0.1117,  0.4155,  0.1389, -0.4171,
          0.6288,  0.5829,  0.0657,  0.2887],
        [ 0.5492,  0.6238, -0.4663,  0.3776, -0.1117,  0.4155,  0.1389, -0.4171,
          0.6288,  0.5829,  0.0657,  0.2887]], grad_fn=<AddBackward0>) torch.Size([2, 12])
```

Please check the cell below to notice the following:

1. `x` above was created with the shape 2x4, and in the forward pass, it gets manipulated into a 2x12 tensor. This last dimension is explicit, while the first is called the batch dimmension, and only exists on data (a.k.a. activations). The output shape can be seen in the print statement from y.shape
2. You can view the shape of a tensor by using `.shape`, this is a very helpful trick for debugging tensor shape errors
3. In the output, there's a `grad_fn` component, this is the hook created by the forward trace to be used in back-propagation via automatic differentiation. The function name is `AddBackward`, because the last operation performed was `h1+h2`.

We might not always want to trace the compute graph though, such as during inference. In such cases, you can use the `torch.no_grad()` context manager.

```
# you can perform a forward pass by first creating a tensor to send through
x = torch.zeros(2,4)
# then you call the module (this invokes MyModule.forward() )
with torch.no_grad():
  y = module(x)

# then you can print the result and shape
```

```
print(y, y.shape)
# notice how the grad_fn is no longer part of the output tensor, that's because not_grad() disables the graph generation
```

```
tensor([[ 0.5492,  0.6238, -0.4663,  0.3776, -0.1117,  0.4155,  0.1389, -0.4171,
          0.6288,  0.5829,  0.0657,  0.2887],
        [ 0.5492,  0.6238, -0.4663,  0.3776, -0.1117,  0.4155,  0.1389, -0.4171,
          0.6288,  0.5829,  0.0657,  0.2887]]) torch.Size([2, 12])
```

Aside from passing a tensor through a model with the `no_grad()` context, you can also detach a tensor from the compute graph by calling `.detach()`. This will effectively make a copy of the original tensor, which allows it to be converted to numpy and visualized with matplotlib.

Note: Tensors with a `grad_fn` property cannot be plotted and must first be detached.

Multi-Layer-Perceptron (MLP) Prediction of MNIST

With some basics out of the way, let's create a MLP for training MNIST. You can start by defining a simple torch model.

```
# Define the MLP model
class MLP(nn.Module):
    # define the constructor for the network
    def __init__(self):
        super().__init__()
        # the input projection layer - projects into d=128
        self.fc1 = nn.Linear(28*28, 128)
        # the first hidden layer - compresses into d=64
        self.fc2 = nn.Linear(128, 64)
        # the final output layer - splits into 10 classes (digits 0-9)
        self.fc3 = nn.Linear(64, 10)

    # define the forward pass compute graph
    def forward(self, x):
        # x is of shape BxHxW

        # we first need to unroll the 2D image using view
        # we set the first dim to be -1 meanining "everything else", the reason being that x is of shape BxHxW, where B is the batch dir
        # we want to maintain different tensors for each training sample in the batch, which means the output should be of shape BxF whe
        x = x.view(-1, 28*28)
        # x is of shape Bx784

        # project-in and apply a non-linearity (ReLU activation function)
        x = torch.relu(self.fc1(x))
        # x is of shape Bx128

        # middle-layer and apply a non-linearity (ReLU activation function)
        x = torch.relu(self.fc2(x))
        # x is of shape Bx64

        # project out into the 10 classes
        x = self.fc3(x)
        # x is of shape Bx10
        return x
```

Before you can begin training, you have to do a little boiler-plate to load the dataset. From the previous assignment, you saw how a hosted dataset can be loaded with TensorFlow. With pytorch it's a little more complicated, as you need to manually condition the input data.

```
# define a transformation for the input images. This uses torchvision.transforms, and .Compose will act similarly to nn.Sequential
transform = transforms.Compose([
    transforms.ToTensor(), # first convert to a torch tensor
    transforms.Normalize((0.1307,), (0.3081,)) # then normalize the input
])

# let's download the train and test datasets, applying the above transform - this will get saved locally into ./data, which is in the Co
train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)

# we need to set the mini-batch (commonly referred to as "batch"), for now we can use 64
batch_size = 64

# then we need to create a dataloader for the train dataset, and we will also create one for the test dataset to evaluate performance
# additionally, we will set the batch size in the dataloader
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# the torch dataloaders allow us to access the __getitem__ method, which returns a tuple of (data, label)
# additionally, the dataloader will pre-colate the training samples into the given batch_size
```

Inspect the first element of the test_loader, and verify both the tensor shapes and data types. You can check the data-type with `.dtype`

Question 1

Edit the cell below to print out the first element shapes, dtype, and identify which is the training sample and which is the training label.

```
# Get the first item
first_item = next(iter(test_loader))

# Get the first batch of data and labels from the test_loader
data_iter = iter(test_loader)
data, labels = next(data_iter)

# Print the shapes and data types
print(f"Data shape: {data.shape}")  # Should be [batch_size, channels, height, width]
print(f"Data dtype: {data.dtype}")
print(f"Labels shape: {labels.shape}")  # Should be [batch_size]
print(f"Labels dtype: {labels.dtype}")

# print out the element shapes, dtype, and identify which is the training sample and which is the training label
# MNIST is a supervised learning task
```

```
Data shape: torch.Size([64, 1, 28, 28])
Data dtype: torch.float32
Labels shape: torch.Size([64])
Labels dtype: torch.int64
```

Now that we have the dataset loaded, we can instantiate the MLP model, the loss (or criterion function), and the optimizer for training.

```
# create the model
model = MLP()

# you can print the model as well, but notice how the activation functions are missing. This is because they were called in the forward
# and not declared in the constructor
print(model)

# you can also count the model parameters
param_count = sum([p.numel() for p in model.parameters()])
print(f"Model has {param_count:,} trainable parameters")

# for a critereon (loss) function, you will use Cross-Entropy Loss. This is the most common criterion used for multi-class prediction,
# and is also used by tokenized transformer models it takes in an un-normalized probability distribution (i.e. without softmax) over
# N classes (in our case, 10 classes with MNIST). This distribution is then compared to an integer label which is < N.
# For MNIST, the prediction might be [-0.0056, -0.2044,  1.1726,  0.0859,  1.8443, -0.9627,  0.9785, -1.0752, 1.1376,  1.8220], with the
# Cross-entropy can be thought of as finding the difference between the predicted distribution and the one-hot distribution

criterion = nn.CrossEntropyLoss()

# then you can instantiate the optimizer. You will use Stochastic Gradient Descent (SGD), and can set the learning rate to 0.1 with a mo
# factor of 0.5. the first input to the optimizer is the list of model parameters, which is obtained by calling .parameters() on the mod
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

```
MLP(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
)
Model has 109,386 trainable parameters
```

Finally, you can define a training, and test loop

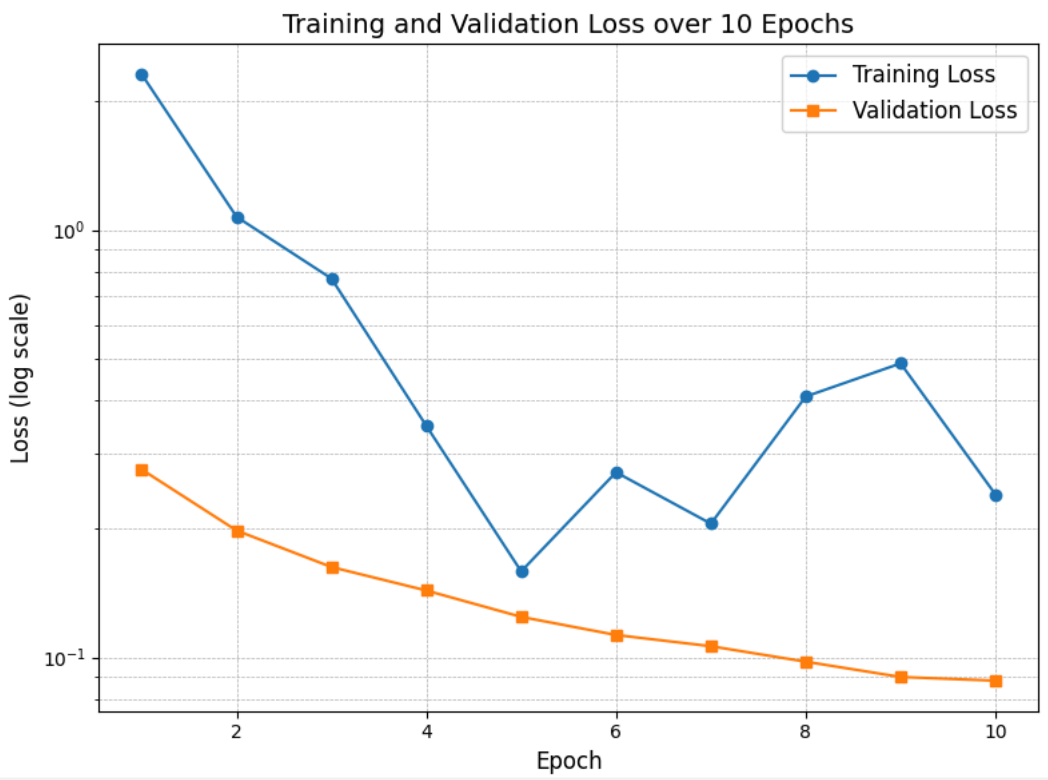
```
# create an array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0


# declare the train function
def cpu_train(epoch, train_losses, steps, current_step):

    # set the model in training mode - this doesn't do anything for us right now, but it is good practiced and needed with other layers
    # batch norm and dropout
    model.train()

    # Create tqdm progress bar to help keep track of the training progress
    pbar = tqdm(enumerate(train_loader), total=len(train_loader))

    # loop over the dataset. Recall what comes out of the data loader, and then by wrapping that with enumerate() we get an index into t
    # iterator list which we will call batch_idx
    for batch_idx, (data, target) in pbar:

        # during training, the first step is to zero all of the gradients through the optimizer
        # this resets the state so that we can begin back propogation with the updated parameters
        optimizer.zero_grad()

        # then you can apply a forward pass, which includes evaluating the loss (criterion)
        output = model(data)
        loss = criterion(output, target)

        # given that you want to minimize the loss, you need to call .backward() on the result, which invokes the grad_fn property
        loss.backward()

        # the backward step will automatically differentiate the model and apply a gradient property to each of the parameters in the ne
        # so then all you have to do is call optimizer.step() to apply the gradients to the current parameters
        optimizer.step()

        # increment the step count
        current_step += 1

        # you should add some output to the progress bar so that you know which epoch you are training, and what the current loss is
        if batch_idx % 100 == 0:

            # append the last loss value
            train_losses.append(loss.item())
            steps.append(current_step)

            desc = (f'Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)}'
                    f' ({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item():.6f}')
            pbar.set_description(desc)

    return current_step

# declare a test function, this will help you evaluate the model progress on a dataset which is different from the training dataset
# doing so prevents cross-contamination and misleading results due to overfitting
def cpu_test(test_losses, test_accuracy, steps, current_step):

    # put the model into eval mode, this again does not currently do anything for you, but it is needed with other layers like batch_nor
    # and dropout
    model.eval()
    test_loss = 0
    correct = 0

    # Create tqdm progress bar
    pbar = tqdm(test_loader, total=len(test_loader), desc="Testing...")

    # since you are not training the model, and do not need back-propagation, you can use a no_grad() context
    with torch.no_grad():
        # iterate over the test set
        for data, target in pbar:
            # like with training, run a forward pass through the model and evaluate the criterion
            output = model(data)
            test_loss += criterion(output, target).item() # you are using .item() to get the loss value rather than the tensor itself

            # you can also check the accuracy by sampling the output - you can use greedy sampling which is argmax (maximum probability)
            # in general, you would want to normalize the logits first (the un-normalized output of the model), which is done via .softm
            # however, argmax is taking the maximum value, which will be the same index for the normalized and un-normalized distributio
            # so we can skip a step and take argmax directly
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader)

    # append the final test loss
    test_losses.append(test_loss)
    test_accuracy.append(correct/len(test_loader.dataset))
    steps.append(current_step)

    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy: {correct}/{len(test_loader.dataset)}'
          f' ({100. * correct / len(test_loader.dataset):.0f}%)\n')



# train for 10 epochs
for epoch in range(0, 10):
    current_step = cpu_train(current_epoch, train_losses, train_steps, current_step)
    cpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1
```

```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.240575: 100%|██████████| 938/938 [00:14<00:00, 64.35it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.32it/s]

Test set: Average loss: 0.2752, Accuracy: 9205/10000 (92%)

Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.328604: 100%|██████████| 938/938 [00:14<00:00, 66.62it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 71.02it/s]

Test set: Average loss: 0.1981, Accuracy: 9424/10000 (94%)

Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.071352: 100%|██████████| 938/938 [00:14<00:00, 64.08it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.60it/s]

Test set: Average loss: 0.1627, Accuracy: 9504/10000 (95%)

Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.386224: 100%|██████████| 938/938 [00:14<00:00, 63.41it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 82.59it/s]

Test set: Average loss: 0.1434, Accuracy: 9566/10000 (96%)

Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.087055: 100%|██████████| 938/938 [00:13<00:00, 68.53it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 81.56it/s]

Test set: Average loss: 0.1244, Accuracy: 9618/10000 (96%)

Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.030143: 100%|██████████| 938/938 [00:14<00:00, 62.95it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 66.80it/s]

Test set: Average loss: 0.1128, Accuracy: 9639/10000 (96%)

Train Epoch: 6 [57600/60000 (96%)]      Loss: 0.106783: 100%|██████████| 938/938 [00:13<00:00, 69.38it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.49it/s]

Test set: Average loss: 0.1062, Accuracy: 9674/10000 (97%)

Train Epoch: 7 [57600/60000 (96%)]      Loss: 0.105279: 100%|██████████| 938/938 [00:13<00:00, 69.58it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 59.01it/s]

Test set: Average loss: 0.0977, Accuracy: 9682/10000 (97%)

Train Epoch: 8 [57600/60000 (96%)]      Loss: 0.137311: 100%|██████████| 938/938 [00:13<00:00, 70.34it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 39.86it/s]

Test set: Average loss: 0.0900, Accuracy: 9706/10000 (97%)

Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.175112: 100%|██████████| 938/938 [00:13<00:00, 68.12it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 62.95it/s]
Test set: Average loss: 0.0882, Accuracy: 9712/10000 (97%)
```

Question 2

Using the skills you acquired in the previous assignment edit the cell below to use matplotlib to visualize the loss for training and validation for the first 10 epochs. They should be plotted on the same graph, labeled, and use a log-scale on the y-axis.

```
# visualize the losses for the first 10 epochs
# Plot training and validation losses
epochs = list(range(1, 11))  # First 10 epochs

plt.figure(figsize=(8, 6))
plt.plot(epochs, train_losses[:10], label='Training Loss', marker='o')
plt.plot(epochs, test_losses[:10], label='Validation Loss', marker='s')

# Customize the plot
plt.yscale('log')  # Use log scale for the y-axis
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Loss (log scale)', fontsize=12)
plt.title('Training and Validation Loss over 10 Epochs', fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

# Show the plot
plt.tight_layout()
plt.show()
```



Question 3

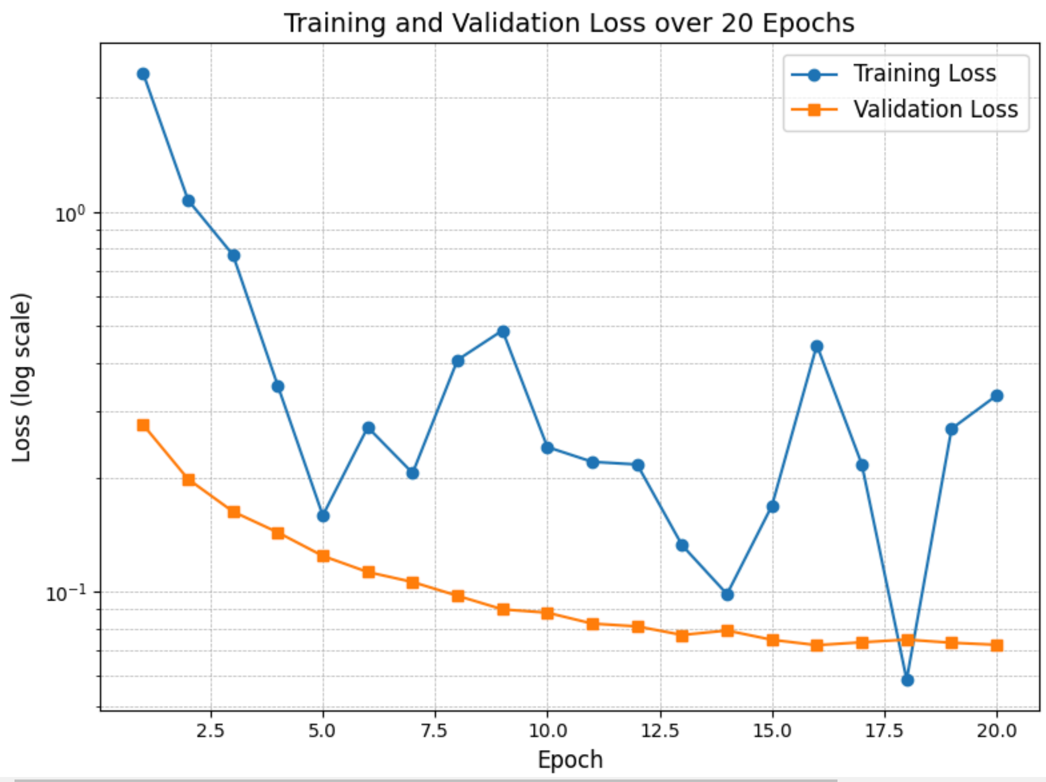
The model may be able to train for a bit longer. Edit the cell below to modify the previous training code to also report the time per epoch and the time for 10 epochs with testing. You can use `time.time()` to get the current time in seconds. Then run the model for another 10 epochs, printing out the execution time at the end, and replot the loss functions with the extra 10 epochs below.

```
# visualize the losses for 20 epochs
# Assuming train_losses, test_losses, train_steps, test_steps, test_accuracy are already defined
# Resetting epoch count for clarity
current_epoch = 0

# Store total start time
total_start_time = time.time()

# Train for 10 epochs and measure time
for epoch in range(10):
    epoch_start_time = time.time()
    current_step = cpu_train(current_epoch, train_losses, train_steps, current_step)
    cpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1
    epoch_end_time = time.time()
    print(f"Epoch {epoch + 1} completed in {epoch_end_time - epoch_start_time:.2f} seconds")

# Print total time taken for the first 10 epochs
total_end_time = time.time()
print(f"Total time for 10 epochs: {total_end_time - total_start_time:.2f} seconds")

# Train for another 10 epochs
extra_start_time = time.time()

for epoch in range(10):
    epoch_start_time = time.time()
    current_step = cpu_train(current_epoch, train_losses, train_steps, current_step)
    cpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1
    epoch_end_time = time.time()
    print(f"Epoch {epoch + 11} completed in {epoch_end_time - epoch_start_time:.2f} seconds")

extra_end_time = time.time()
print(f"Total time for the additional 10 epochs: {extra_end_time - extra_start_time:.2f} seconds")
print(f"Overall training time for 20 epochs: {extra_end_time - total_start_time:.2f} seconds")

# Plot the updated loss functions for 20 epochs
epochs = list(range(1, 21))  # 1 to 20 epochs

plt.figure(figsize=(8, 6))
plt.plot(epochs, train_losses[:20], label='Training Loss', marker='o')
plt.plot(epochs, test_losses[:20], label='Validation Loss', marker='s')
```

```
# Customize the plot
plt.yscale('log')  # Use log scale for the y-axis
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Loss (log scale)', fontsize=12)
plt.title('Training and Validation Loss over 20 Epochs', fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

# Show the plot
plt.tight_layout()
plt.show()
```

```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.060721: 100%|██████████| 938/938 [00:13<00:00, 72.05it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 58.55it/s]

Test set: Average loss: 0.0825, Accuracy: 9731/10000 (97%)

Epoch 1 completed in 15.72 seconds
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.022600: 100%|██████████| 938/938 [00:13<00:00, 69.96it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 78.47it/s]

Test set: Average loss: 0.0812, Accuracy: 9735/10000 (97%)

Epoch 2 completed in 15.42 seconds
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.003945: 100%|██████████| 938/938 [00:13<00:00, 68.54it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 82.60it/s]

Test set: Average loss: 0.0771, Accuracy: 9765/10000 (98%)

Epoch 3 completed in 15.60 seconds
Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.009724: 100%|██████████| 938/938 [00:13<00:00, 69.74it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.88it/s]

Test set: Average loss: 0.0792, Accuracy: 9764/10000 (98%)

Epoch 4 completed in 15.41 seconds
Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.038439: 100%|██████████| 938/938 [00:14<00:00, 66.47it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 51.51it/s]

Test set: Average loss: 0.0748, Accuracy: 9770/10000 (98%)

Epoch 5 completed in 17.18 seconds
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.032297: 100%|██████████| 938/938 [00:13<00:00, 68.85it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 81.54it/s]

Test set: Average loss: 0.0724, Accuracy: 9784/10000 (98%)

Epoch 6 completed in 15.56 seconds
Train Epoch: 6 [57600/60000 (96%)]      Loss: 0.039517: 100%|██████████| 938/938 [00:14<00:00, 66.12it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 78.96it/s]

Test set: Average loss: 0.0737, Accuracy: 9764/10000 (98%)

Epoch 7 completed in 16.19 seconds
Train Epoch: 7 [57600/60000 (96%)]      Loss: 0.066862: 100%|██████████| 938/938 [00:13<00:00, 69.83it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.52it/s]

Test set: Average loss: 0.0749, Accuracy: 9767/10000 (98%)

Epoch 8 completed in 15.50 seconds
Train Epoch: 8 [57600/60000 (96%)]      Loss: 0.029261: 100%|██████████| 938/938 [00:13<00:00, 67.83it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 50.73it/s]

Test set: Average loss: 0.0735, Accuracy: 9774/10000 (98%)

Epoch 9 completed in 16.94 seconds
Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.017397: 100%|██████████| 938/938 [00:13<00:00, 70.03it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.34it/s]

Test set: Average loss: 0.0726, Accuracy: 9777/10000 (98%)

Epoch 10 completed in 15.36 seconds
Total time for 10 epochs: 158.88 seconds
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.047573: 100%|██████████| 938/938 [00:13<00:00, 70.00it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 82.12it/s]

Test set: Average loss: 0.0710, Accuracy: 9782/10000 (98%)

Epoch 11 completed in 15.33 seconds
Train Epoch: 11 [57600/60000 (96%)]     Loss: 0.026854: 100%|██████████| 938/938 [00:14<00:00, 65.24it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 59.75it/s]

Test set: Average loss: 0.0718, Accuracy: 9788/10000 (98%)

Epoch 12 completed in 17.02 seconds
Train Epoch: 12 [57600/60000 (96%)]     Loss: 0.063858: 100%|██████████| 938/938 [00:13<00:00, 68.74it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 82.45it/s]

Test set: Average loss: 0.0707, Accuracy: 9791/10000 (98%)

Epoch 13 completed in 15.57 seconds
Train Epoch: 13 [57600/60000 (96%)]     Loss: 0.022578: 100%|██████████| 938/938 [00:13<00:00, 68.13it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 81.45it/s]

Test set: Average loss: 0.0724, Accuracy: 9786/10000 (98%)

Epoch 14 completed in 15.71 seconds
Train Epoch: 14 [57600/60000 (96%)]     Loss: 0.010266: 100%|██████████| 938/938 [00:13<00:00, 70.79it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 83.80it/s]

Test set: Average loss: 0.0701, Accuracy: 9791/10000 (98%)
```

```
Epoch 15 completed in 15.14 seconds
Train Epoch: 15 [57600/60000 (96%)]     Loss: 0.022810: 100%|██████████| 938/938 [00:13<00:00, 67.22it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 57.09it/s]

Test set: Average loss: 0.0733, Accuracy: 9788/10000 (98%)

Epoch 16 completed in 16.72 seconds
Train Epoch: 16 [57600/60000 (96%)]     Loss: 0.003723: 100%|██████████| 938/938 [00:13<00:00, 68.88it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 81.88it/s]

Test set: Average loss: 0.0712, Accuracy: 9788/10000 (98%)

Epoch 17 completed in 15.55 seconds
Train Epoch: 17 [57600/60000 (96%)]     Loss: 0.006635: 100%|██████████| 938/938 [00:13<00:00, 67.40it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.95it/s]

Test set: Average loss: 0.0738, Accuracy: 9784/10000 (98%)

Epoch 18 completed in 15.87 seconds
Train Epoch: 18 [57600/60000 (96%)]     Loss: 0.008495: 100%|██████████| 938/938 [00:13<00:00, 69.81it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 80.05it/s]

Test set: Average loss: 0.0720, Accuracy: 9792/10000 (98%)

Epoch 19 completed in 15.41 seconds
Train Epoch: 19 [57600/60000 (96%)]     Loss: 0.006370: 100%|██████████| 938/938 [00:13<00:00, 69.99it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 56.17it/s]

Test set: Average loss: 0.0747, Accuracy: 9787/10000 (98%)

Epoch 20 completed in 16.21 seconds
Total time for the additional 10 epochs: 158.53 seconds
Overall training time for 20 epochs: 317.41 seconds
```



Question 4

Make an observation from the above plot. Do the test and train loss curves indicate that the model should train longer to improve accuracy? Or does it indicate that 20 epochs is too long? Edit the cell below to answer these questions.

From the graph of "training loss" we can observe the graph has not yet reached a "plateau" or did not yet saturate, indicating that the model is still learning. Hence increasing the no of training epochs will improve training accuracy. Furthermore, the large gap between the "training loss" and the "validation loss" may indicate overfitting. Since the model has not "converged" and there is a large gap between both the plots (suggesting overfitting), decresing the no of epochs may inprove convergence.

Double-click (or enter) to edit

Moving to the GPU

Now that you have a model trained on the CPU, let's finally utilize the T4 GPU that we requested for this instance.

Using a GPU with torch is relatively simple, but has a few gotchas. Torch abstracts away most of the CUDA runtime API, but has a few hold-over concepts such as moving data between devices. Additionally, since the GPU is treated as a device separate from the CPU, you cannot combine CPU and GPU based tensors in the same operation. Doing so will result in a device mismatch error. If this occurs, check where the tensors are located (you can always print `.device` on a tensor), and make sure they have been properly moved to the correct device.

You will start by creating a new model, optimizer, and criterion (not really necessary in this case since you already did this above but it's better for clarity and completeness). However, one change that you'll make is moving the model to the GPU first. This can be done by calling `.cuda()` in general, or `.to("cuda")` to be more explicit. In general specific GPU devices can be targetted such as `.to("cuda:0")` for the first GPU (index 0), etc., but since there is only one GPU in Colab this is not necessary in this case.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)  # Input layer (flattened 28x28 image)
        self.fc2 = nn.Linear(128, 64)       # Hidden layer
        self.fc3 = nn.Linear(64, 10)        # Output layer (10 classes)

    def forward(self, x):
        x = x.view(-1, 28 * 28)  # Flatten the image
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)  # No activation here (CrossEntropyLoss expects raw logits)
        return x

# Instantiate the model
model = MLP()

# Move the model to GPU (if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

print("Model and optimizer are set up!")
```

```
Model and optimizer are set up!
```

```
# create a new array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0
```

Now, copy your previous training code with the timing parameters below. It needs to be slightly modified to move everything to the GPU.

Before the line `output = model(data)`, add:

```
data = data.cuda()
target = target.cuda()
```

Note that this is needed in both the train and test functions.

Question 5

Please edit the cell below to show the new GPU train and test fucntions.

```
import torch
import time
import matplotlib.pyplot as plt

# Assuming train_losses, test_losses, train_steps, test_steps, test_accuracy are already defined
current_epoch = 0

# Move model to GPU (if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Store total start time
total_start_time = time.time()

# Train for 10 epochs and measure time
for epoch in range(10):
    epoch_start_time = time.time()

    # Training
    for batch_idx, (data, target) in enumerate(train_loader):  # Assuming train_loader is defined
        data, target = data.to(device), target.to(device)  # Move data & target to GPU
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

    # Validation
    with torch.no_grad():
        test_loss = 0
        correct = 0
        for data, target in test_loader:  # Assuming test_loader is defined
            data, target = data.to(device), target.to(device)  # Move data & target to GPU
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

        test_loss /= len(test_loader.dataset)
        test_losses.append(test_loss)
        test_accuracy.append(100. * correct / len(test_loader.dataset))

    current_epoch += 1
    epoch_end_time = time.time()
    print(f"Epoch {epoch + 1} completed in {epoch_end_time - epoch_start_time:.2f} seconds")

# Print total time taken for the first 10 epochs
total_end_time = time.time()
print(f"Total time for 10 epochs: {total_end_time - total_start_time:.2f} seconds")

# Train for another 10 epochs
extra_start_time = time.time()

for epoch in range(10):
    epoch_start_time = time.time()

    # Training
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)  # Move data & target to GPU
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())

    # Validation
    with torch.no_grad():
        test_loss = 0
        correct = 0
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)  # Move data & target to GPU
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

        test_loss /= len(test_loader.dataset)
        test_losses.append(test_loss)
        test_accuracy.append(100. * correct / len(test_loader.dataset))

    current_epoch += 1
    epoch_end_time = time.time()
    print(f"Epoch {epoch + 11} completed in {epoch_end_time - epoch_start_time:.2f} seconds")

extra_end_time = time.time()
print(f"Total time for the additional 10 epochs: {extra_end_time - extra_start_time:.2f} seconds")
print(f"Overall training time for 20 epochs: {extra_end_time - total_start_time:.2f} seconds")

# Plot the updated loss functions for 20 epochs
epochs = list(range(1, 21))  # 1 to 20 epochs

plt.figure(figsize=(8, 6))
plt.plot(epochs, train_losses[:20], label='Training Loss', marker='o')
plt.plot(epochs, test_losses[:20], label='Validation Loss', marker='s')

# Customize the plot
plt.yscale('log')  # Use log scale for the y-axis
plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Loss (log scale)', fontsize=12)
plt.title('Training and Validation Loss over 20 Epochs', fontsize=14)
plt.legend(fontsize=12)
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

# Show the plot
plt.tight_layout()
plt.show()
```

```
Epoch 1 completed in 14.35 seconds
Epoch 2 completed in 14.67 seconds
Epoch 3 completed in 14.09 seconds
Epoch 4 completed in 14.11 seconds
Epoch 5 completed in 14.13 seconds
Epoch 6 completed in 13.95 seconds
Epoch 7 completed in 14.02 seconds
Epoch 8 completed in 14.36 seconds
Epoch 9 completed in 14.43 seconds
Epoch 10 completed in 14.15 seconds
Total time for 10 epochs: 142.25 seconds
Epoch 11 completed in 14.11 seconds
Epoch 12 completed in 14.12 seconds
Epoch 13 completed in 14.03 seconds
Epoch 14 completed in 14.26 seconds
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-29-d43d4fb4ece2> in <cell line: 0>()
     57 
     58     # Training
---> 59     for batch_idx, (data, target) in enumerate(train_loader):
     60         data, target = data.to(device), target.to(device)  # Move data & target to GPU
     61         optimizer.zero_grad()

                                   10 frames
/usr/local/lib/python3.11/dist-packages/torchvision/transforms/_functional_tensor.py in normalize(tensor, mean, std, inplace)
    918 
    919     dtype = tensor.dtype
--> 920     mean = torch.as_tensor(mean, dtype=dtype, device=tensor.device)
    921     std = torch.as_tensor(std, dtype=dtype, device=tensor.device)
    922     if (std == 0).any():

KeyboardInterrupt: 
```

```
import torch
import time

# Move model to GPU (if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Function for training on GPU
def train_gpu(model, train_loader, optimizer, criterion, train_losses):
    model.train()  # Set model to training mode
    total_loss = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)  # Move to GPU
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    train_losses.append(avg_loss)  # Store average training loss

# Function for testing on GPU
def test_gpu(model, test_loader, criterion, test_losses, test_accuracy):
    model.eval()  # Set model to evaluation mode
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)  # Move to GPU
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)
    test_accuracy.append(100. * correct / len(test_loader.dataset))

# 🔹 Training Loop for 10 Epochs
num_epochs = 10
train_losses, test_losses, test_accuracy = [], [], []

start_time = time.time()

for epoch in range(1, num_epochs + 1):
    epoch_start = time.time()

    train_gpu(model, train_loader, optimizer, criterion, train_losses)
    test_gpu(model, test_loader, criterion, test_losses, test_accuracy)

    epoch_end = time.time()
    print(f"Epoch {epoch}/{num_epochs} - Time: {epoch_end - epoch_start:.2f}s")

end_time = time.time()
print(f"Total training time for {num_epochs} epochs: {end_time - start_time:.2f} seconds")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-57d613641138> in <cell line: 0>()
      4 # Move model to GPU (if available)
      5 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
----> 6 model.to(device)
      7
      8 # Function for training on GPU

NameError: name 'model' is not defined
```

Question 6

Is training faster now that it is on a GPU? Is the speedup what you would expect? Why or why not? Edit the cell below to answer.

Double-click (or enter) to edit

Another Model Type: CNN

Until now you have trained a simple MLP for MNIST classification, however, MLPs are not a particularly good for images.

Firstly, using a MLP will require that all images have the same size and shape, since they are unrolled in the input.

Secondly, in general images can make use of translation invariance (a type of data symmetry), but this cannot but leveraged with a MLP.

For these reasons, a convolutional network is more appropriate, as it will pass kernels over the 2D image, removing the requirement for a fixed image size and leveraging the translation invariance of the 2D images.

Let's define a simple CNN below.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the CNN model
class CNN(nn.Module):
    # Define the constructor for the network
    def __init__(self):
        super().__init__()

        # Convolutional network
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),  # Input layer (28x28 -> 28x28)
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1), # Downsampling (28x28 -> 14x14)
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1), # Downsampling (14x14 -> 7x7)
            nn.ReLU(),
            nn.AdaptiveMaxPool2d(1),  # Pooling layer (7x7 -> 1x1)
        )

        # Prediction head (MLP)
        self.head = nn.Sequential(
            nn.Linear(128, 64),  # 128 -> 64
            nn.ReLU(),
            nn.Linear(64, 10)    # 64 -> 10 (for digit classification)
        )

    # Forward pass
    def forward(self, x):
        x = self.net(x)  # Pass through CNN
        x = x.view(x.size(0), -1)  # Flatten (Bx128x1x1 → Bx128)
        x = self.head(x)  # Pass through MLP head
        return x

# Instantiate the model
model = CNN()
print(model)  # Print model architecture
```

```
CNN(
  (net): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (5): ReLU()
    (6): AdaptiveMaxPool2d(output_size=1)
  )
  (head): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

```
# create the model
model = CNN()

# print the model and the parameter count
print(model)
param_count = sum([p.numel() for p in model.parameters()])
print(f"Model has {param_count:,} trainable parameters")

# the loss function
criterion = nn.CrossEntropyLoss()

# then you can intantiate the optimizer. You will use Stochastic Gradient Descent (SGD), and can set the learning rate to 0.1 with a
# momentum factor of 0.5
# the first input to the optimizer is the list of model parameters, which is obtained by calling .parameters() on the model object
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

```
CNN(
  (net): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
```

```
        (3): ReLU()
        (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (5): ReLU()
        (6): AdaptiveMaxPool2d(output_size=1)
      )
      (head): Sequential(
        (0): Linear(in_features=128, out_features=64, bias=True)
        (1): ReLU()
        (2): Linear(in_features=64, out_features=10, bias=True)
      )
    )
    Model has 101,578 trainable parameters
```

Question 7

Notice that this model now has fewer parameters than the MLP. Let's see how it trains.

Using the previous code to train on the CPU with timing, edit the cell below to execute 2 epochs of training.

```
import time

# create a new array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0

# Assuming train_loader and test_loader are already defined

# Define the CPU training function
def cpu_train(epoch, model, train_loader, optimizer, criterion):
    model.train()  # Set model to training mode
    running_loss = 0.0
    for i, (inputs, labels) in enumerate(train_loader, 0):
        inputs, labels = inputs.to('cpu'), labels.to('cpu')  # Move data to CPU
        optimizer.zero_grad()  # Zero the gradients
        outputs = model(inputs)  # Forward pass
        loss = criterion(outputs, labels)  # Calculate loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update weights
        running_loss += loss.item()
        current_step += 1
        # Log every 100th batch
        if current_step % 100 == 0:
            print(f"Epoch [{epoch+1}], Step [{current_step}], Loss: {running_loss / 100:.4f}")
            running_loss = 0.0

# Define the CPU testing function
def cpu_test(model, test_loader, criterion):
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    test_loss = 0.0
    with torch.no_grad():  # Disable gradient calculation for testing
        for inputs, labels in test_loader:
            inputs, labels = inputs.to('cpu'), labels.to('cpu')
            outputs = model(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Calculate loss
            test_loss += loss.item()
            _, predicted = torch.max(outputs, 1)  # Get the predicted class
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    avg_test_loss = test_loss / len(test_loader)
    accuracy = 100 * correct / total
    test_losses.append(avg_test_loss)
    test_accuracy.append(accuracy)
    print(f"Test Loss: {avg_test_loss:.4f}, Accuracy: {accuracy:.2f}%")

# Train for 2 epochs on CPU
start_time = time.time()

for epoch in range(2):  # Train for 2 epochs
    epoch_start = time.time()
    cpu_train(epoch, model, train_loader, optimizer, criterion)  # Train the model
    cpu_test(model, test_loader, criterion)  # Test the model
    epoch_end = time.time()
    print(f"Epoch {epoch + 1} training time: {epoch_end - epoch_start:.2f}s")

end_time = time.time()
end_time = time.time()
print(f"Total training time for 2 epochs: {end_time - start_time:.2f} seconds")
```

...

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.utils.data as data
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import time

# ✅ Define batch size
batch_size = 64

# ✅ Define data transformations (convert images to tensors & normalize)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# ✅ Download and create MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

# ✅ Create DataLoaders for training and testing
train_loader = data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# ✅ Define CNN Model
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.AdaptiveMaxPool2d(1),
        )
        self.head = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.net(x)
        x = x.view(x.size(0), -1)
        x = self.head(x)
        return x

# ✅ Define CPU Training Function
def cpu_train(epoch, model, train_loader, optimizer, criterion):
    global current_step
    model.train()
    running_loss = 0.0

    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        current_step += 1

    avg_loss = running_loss / len(train_loader)
```

```
        train_losses.append(avg_loss)
        train_steps.append(current_step)
        print(f"Epoch {epoch}: Training Loss = {avg_loss:.4f}")

    # ✅ Define CPU Testing Function
    def cpu_test(model, test_loader, criterion):
        model.eval()
        test_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for data, target in test_loader:
                output = model(data)
                loss = criterion(output, target)
                test_loss += loss.item()
                pred = output.argmax(dim=1, keepdim=True)
                correct += pred.eq(target.view_as(pred)).sum().item()
                total += target.size(0)

        avg_loss = test_loss / len(test_loader)
        accuracy = 100.0 * correct / total
        test_losses.append(avg_loss)
        test_accuracy.append(accuracy)
        test_steps.append(current_step)
        print(f"Test Loss = {avg_loss:.4f}, Accuracy = {accuracy:.2f}%")

    # ✅ Training Setup
    device = torch.device("cpu")  # Using CPU
    model = CNN().to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

    # ✅ Initialize Logging Variables
    train_losses, test_losses, test_accuracy = [], [], []
    train_steps, test_steps = [], []
    current_step = 0
    current_epoch = 0

    # ✅ Train for 2 Epochs
    num_epochs = 2
    start_time = time.time()

    for epoch in range(1, num_epochs + 1):
        epoch_start = time.time()
        cpu_train(epoch, model, train_loader, optimizer, criterion)
        cpu_test(model, test_loader, criterion)
        epoch_end = time.time()
        print(f"Epoch {epoch}/{num_epochs} completed in {epoch_end - epoch_start:.2f} seconds")

    end_time = time.time()
    print(f"Total training time for {num_epochs} epochs: {end_time - start_time:.2f} seconds")
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9.91M/9.91M [00:02<00:00, 4.52MB/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28.9k/28.9k [00:00<00:00, 113kB/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz
100%|██████████| 1.65M/1.65M [00:01<00:00, 1.26MB/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>
```

```
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4.54k/4.54k [00:00<00:00, 5.40MB/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Epoch 1: Training Loss = 2.2313
Test Loss = 1.7433, Accuracy = 39.77%
Epoch 1/2 completed in 67.74 seconds
Epoch 2: Training Loss = 0.6740
Test Loss = 0.2723, Accuracy = 91.62%
Epoch 2/2 completed in 74.46 seconds
Total training time for 2 epochs: 142.20 seconds
```

Question 8

Now, let's move the model to the GPU and try training for 2 epochs there.

```
import torch
import torch.nn as nn
import torch.optim as optim

# ✅ Define the CNN Model
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.AdaptiveMaxPool2d(1),
        )
        self.head = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.net(x)
        x = x.view(x.size(0), -1)
        x = self.head(x)
        return x

# ✅ Instantiate the Model and Move to GPU
model = CNN()
model.cuda()  # Move model to GPU if available

# ✅ Print the Model Summary and Parameter Count
print(model)
param_count = sum(p.numel() for p in model.parameters())
print(f"Model has {param_count:,} trainable parameters")

# ✅ Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

```
CNN(
  (net): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (5): ReLU()
    (6): AdaptiveMaxPool2d(output_size=1)
  )
  (head): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=10, bias=True)
  )
)
Model has 101,578 trainable parameters
```

```
# create a new array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
```

```
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0


import time
import torch
import torch.optim as optim
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define transformations (convert to tensor and normalize)
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize to [-1, 1] range
])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

# Create DataLoader for train and test datasets
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Define the CNN model (make sure you've defined CNN class already)
model = CNN()
model.cuda()

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# Define GPU training and testing functions (as previously defined)
def train_gpu(model, train_loader, optimizer, criterion, train_losses):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for data, target in train_loader:
        data, target = data.cuda(), target.cuda()  # Move data and target to GPU

        optimizer.zero_grad()
        output = model(data)

        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(output, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

    avg_loss = running_loss / len(train_loader)
    accuracy = correct / total
    train_losses.append(avg_loss)
    print(f'Train Loss: {avg_loss:.4f}, Accuracy: {accuracy:.4f}')

def test_gpu(model, test_loader, criterion, test_losses, test_accuracy):
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.cuda(), target.cuda()  # Move data and target to GPU

            output = model(data)
            loss = criterion(output, target)

            running_loss += loss.item()
            _, predicted = torch.max(output, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    avg_loss = running_loss / len(test_loader)
    accuracy = correct / total
    test_losses.append(avg_loss)
    test_accuracy.append(accuracy)
    print(f'Test Loss: {avg_loss:.4f}, Accuracy: {accuracy:.4f}')

# Train and test for 2 epochs on the GPU
train_losses, test_losses, test_accuracy = [], [], []

num_epochs = 2
start_time = time.time()

for epoch in range(1, num_epochs + 1):
    epoch_start = time.time()

    train_gpu(model, train_loader, optimizer, criterion, train_losses)
    test_gpu(model, test_loader, criterion, test_losses, test_accuracy)

    epoch_end = time.time()
    print(f"Epoch {epoch}/{num_epochs} - Time: {epoch_end - epoch_start:.2f}s")

end_time = time.time()
print(f"Total training time for {num_epochs} epochs: {end_time - start_time:.2f} seconds")
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9.91M/9.91M [00:00<00:00, 14.7MB/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28.9k/28.9k [00:00<00:00, 481kB/s]
Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
```

Question 9

How do the CPU and GPU versions compare for the CNN? Is one faster than the other? Why do you think this is, and how does it differ from the MLP? Edit the cell below to answer.

CNN on CPU vs. GPU: GPU (Graphics Processing Unit): Parallelism: GPUs are designed to handle many tasks simultaneously. They are highly optimized for performing large matrix and tensor operations in parallel. CNNs, which involve many convolutions and matrix multiplications, benefit significantly from this parallelism. Speed: When running on a GPU, CNNs tend to perform faster, especially with larger datasets and complex models. The GPU accelerates the processing of multiple data points (images) in parallel, drastically reducing computation time for operations like convolutions, which are computationally expensive. Memory Bandwidth: GPUs have significantly higher memory bandwidth compared to CPUs, which allows them to transfer large amounts of data (e.g., images, feature maps) much more quickly, reducing bottlenecks during training and inference. CPU (Central Processing Unit): Sequential Processing: CPUs are designed for single-threaded tasks and do not handle the massive parallelism required for CNNs as effectively as GPUs. While CPUs can handle CNNs, they will take longer to process due to their lower number of cores and limited vectorized operations. Speed: For smaller models or datasets, a CPU may perform adequately, but as the model size increases (in terms of parameters and layers), the time taken for training and inference becomes much slower compared to a GPU. Lower Memory Bandwidth: CPUs have lower memory bandwidth than GPUs, which limits how quickly data can be transferred for